13. Gull, W.E., and Jenkins, M.A. Recursive data structures and related control mechanisms in APL. Proc. APL76, Ottawa, Canada, 1976, pp. 201–210.
14. Haegi, H.R. Some questions and thoughts concerning arrays of arrays. Proc. SEAS-APL Working Committee, Jan. 1974.
15. Haegi, H.R. The extension of APL to treelike data structures. APL Quote Quad (ACM) *7*, 2 (1976), 8–18.
16. Hoare, C.A.R. Recursive data structures. *Int. J. Comptrs. Syst. Sci. 4*, 2 (1975), 105–132.
17. Iverson, K.E. *A Programming Language.* Wiley, New York, 1962.
18. Landin, P.J. The next 700 programming languages. *Comm. ACM 9*, 3 (March 1966), 157–164.
19. McCarthy, J. LISP 1.5 Programmer's Manual. M.I.T., Cambridge, Mass., 1962.
20. Mercer, R. Extensions of APL to include arrays of arrays: A comparison of three systems. Tech. Rep. Comptng. Ctr., U. of Mass., Amherst, Mass., 1976.
21. More, T. An extension of APL to a theory of arrays. Abstract, 1968. Reprinted in Tech. Rep. 320–3016, IBM Scientific Ctr., Philadelphia, Pa., 1973.
22. More, T. An extension of APL to a theory of arrays. Class notes, Yale U., New Haven, Conn., 1970.
23. More, T. Axioms and theorems for a theory of arrays. *IBM J. Res. Develop. 17*, 3 (1973), 135–175.
24. More, T. Notes on the development of a theory of arrays. Tech. Rep. 320–3016, IBM Scientific Ctr., Philadelphia, Pa., 1973.
25. More, T. Notes on the axioms for a theory of arrays. Tech. Rep. 320–3017, IBM Scientific Ctr., Philadelphia, Pa., 1973.
26. More, T. A theory of arrays with applications to databases. Rep. G320–2106, IBM Scientific Ctr., Cambridge, Mass., 1975.
27. More, T. Types and prototypes in a theory of arrays. Rep. 320–2112, IBM Scientific Ctr., Cambridge, Mass., 1976.
28. More, T. On the composition of array-theoretic operations. Rep. 320–2113, IBM Scientific Ctr., Cambridge, Mass., 1976.
29. Murray, R.C. On tree structured extensions to the APL language. Proc. APL Congress 73, Copenhagen, Denmark, 1973, pp. 333–338.
30. Perlis, A.J. Steps towards an APL compiler—updated. Res. Rep. 24, Comptr. Sci. Dept. Yale U., New Haven, Conn., 1975.
31. Quine, W.V. Unification of universes in set theory. *J. Symbolic Logic 21* (1956), 267–279.
32. Robichaud, L. Trees in APL. Presentation at Queen's APL Workshop, Queen's U., Kingston, Ontario, 1976.
33. Seeds, G. Private communication, 1976.
34. Smith, R.A. The semantics of split were suggested by Bob Smith at the Queen's APL Workshop, Queen's University, Kingston, Ontario, May 1976. One of the referees of the first draft made a similar suggestion.
35. Tennent, R.D. Language design methodologies based on semantic principles. *Acta Informatica 8*, (1977), 97–112.
36. Vasseur, J.P. Extensions of APL operators to tree-like data structures. Proc. APL Congress 73, Copenhagen, Denmark, 1973, pp. 457–464.
37. Wirth, N. On the design of programming languages. Inform. Processing 74, J.L. Rosenfeld, Ed., North-Holland Pub. Co., Amsterdam, 1974, pp. 386–393.

Programming Languages | J. J. Horning Editor

Global Optimization by Suppression of Partial Redundancies

E. Morel and C. Renvoise
CII Honeywell Bull

The elimination of redundant computations and the moving of invariant computations out of loops are often done separately, with invariants moved outward loop by loop. We propose to do both at once and to move each expression directly to the entrance of the outermost loop in which it is invariant. This is done by solving a more general problem, i.e. the elimination of computations performed twice on a given execution path. Such computations are termed partially redundant. Moreover, the algorithm does not require any graphical information or restrictions on the shape of the program graph. Testing this algorithm has shown that its execution cost is nearly linear with the size of the program, and that it leads to a smaller optimizer that requires less execution time.

Key Words and Phrases: optimizer, optimization, compiler, compilation, redundancy elimination, invariant computation elimination, partial redundancy, data flow analysis, Boolean systems

CR Categories: 4.12, 5.21, 5.24

Introduction

The evolution towards high-level programming languages, together with the growing emphasis on reliability and readability, has caused an increase in the distance between the source text and the object code of programs. Since efficiency remains an important goal, it becomes desirable to increase the power of the transformations

This work was supported by the Direction des Recherches et Moyens d'Essais of France.

Authors' address: Compagnie Internationale pour L'informatique Honeywell Bull, Louveciennes 78430, France.

performed by compilers, and in particular, to apply global optimization techniques.

This paper concentrates on the techniques for redundancy elimination and the removal of loop invariants. These transformations decrease the number of run-time computations performed by programs. They have been introduced by Allen and Cocke [2, 4] who gave a formulation based on the notion of intervals. A simpler formulation of these problems, based on the direct resolution of Boolean equations, has been developed by several authors [6, 9, 12].

In the first part of this paper, we present the Boolean approach to global optimization. In the second part, we show that redundancy elimination and loop-invariants removal are aspects of a larger problem, i.e. that of the suppression of partial redundancy. We present an algorithm for their global suppression. This algorithm can be applied to the whole program without any restrictions on the shape of its flowgraph.

2. Elimination of Redundancies and Removal of Invariants

The optimizations described in this paper are applied to a program representation obtained by decomposition of the source text into elementary commands associated with a directed graph. All assignment statements of the source text are split into evaluation of the right-hand side, followed by assignment of the result to the left-hand side. Moreover, if necessary, expressions are split into binary operations.

For instance, the source text instruction:

"$A := A + B + C$;"

results in two expression computations:

$e1 \leftarrow A + B$
$e2 \leftarrow e1 + C$

and one assignment:

$A := e2$.

The nodes of the graph represent the basic blocks. A basic block is a maximal sequence of elementary commands with a single entry point and a single exit point. There is an arc in the graph from the exit point of i to the entry point of j if the block j can be executed immediately after the block i. The set of predecessors and successors of block i are denoted Pred(i) and Succ(i), respectively. The set of all the blocks of the program is denoted B. Paths are taken to be sequences of arcs.

There is a designated entry block and all blocks can be reached from this block. The nodes for which Succ(i) is an empty set are the exit blocks of the program.

Although the notion of loop is not important for our algorithm, we introduce some definitions derived from [11] and [1] in order to simplify certain points of the explanation.

Definition 1. A loop L is a strongly connected set of blocks, i.e.: $\forall i, j \in L$, there is a path in L from i to j.

Definition 2. A block $i \in L$ is an entry block of L if: $\exists j \in \text{Pred}(i) | j \notin L$.

Definition 3. A block $i \in L$ is a repeat block of L if: $\exists j \in \text{Succ}(i) | j$ is an entry block of L.

Definition 4. A backwards edge is an edge (i, j) such that i is a repeat block of L and j is an entry block of L.

Definition 5. A block i is an initialization block of L if: $i \notin L$, and $\exists j \in \text{Succ}(i) | j$ is an entry block of L. If the block i has just one successor, then it is a satisfactory initialization block. Otherwise, a new empty block k is inserted on the edge (i, j) and k becomes the initialization block. In a goto-free language, there is a unique initialization block for every loop.

The optimizations considered involve suppressing or moving some expression computations. In order to determine the feasibility of these transformations, one must identify the mutual interactions of commands located in different blocks. The effect of these interactions can be represented by Boolean properties characterizing each expression with respect to each block.

2.1 Boolean Properties Associated with the Expressions

For each expression and each block, Boolean properties are defined. Some of these properties depend only on the commands of a given block and are termed local. Other properties depend on interactions of different blocks and are termed global.

2.1.1 Local Properties. The local properties are transparency, availability, and anticipability.

Transparency: TRANSP. An expression is said to be "transparent" in a block i if its operands are not modified by the execution of the commands of the block i.

Local Availability: COMP. An expression is said to be "locally available" in a block i if there is at least one computation of the expression in the block i, and if the commands appearing in the block after the last computation of the expression do not modify its operands.

Local Anticipability: ANTLOC. An expression may be locally anticipated in a block i if there is at least one computation of the expression in the block i, and if the commands appearing in the block before the first computation of the expression do not modify its operands.

Local availability of an expression grants that the last computation of this expression in the block will deliver the same result as would a computation of this expression placed at the end of the block. Similarly, local anticipability of an expression grants that the first computation of this expression in the block will deliver the same result as would a computation of this expression placed at the beginning of the block.

2.1.2 Global Properties. The meaning of the previous properties can be extended to a complete program. The availability of an expression at a given point implies that a computation of this expression placed at this point would deliver the same result as the last computation of

this expression made before this point. Similarly, the anticipability of an expression at a given point implies that a computation of this expression placed at this point would deliver the same result as the first computation of this expression made after this point.

The partial availability of an expression at a given point is a weaker property. It means that there is at least one path P leading from the entry point of the program to the point considered, and that computation of the expression inserted at this point would give the same result as the last computation of the expression made on the path P.

These properties qualify program points and are defined whether the expression occurs on the point or not. In practice, we will concentrate on points which are block entries or exits.

For the sake of clarity, only one expression will be considered and subscripts will be used to index the set B. We will use $AVIN_i$, $ANTIN_i$, $PAVIN_i$ to denote, respectively, the global availability, anticipability, and partial availability of the expression on entry of the block i; similarly, $AVOUT_i$, $ANTOUT_i$, $PAVOUT_i$ will be used to denote the same properties on exit of the block i. The relations between global and local properties for all blocks of the graph are expressed in the form of systems of Boolean equations. Boolean conjunctions are denoted . and $\prod$; similarly, + and $\sum$ are used for boolean disjunctions. The operator for Boolean negation is denoted $\neg$. In a Boolean expression, "+" has a weaker precedence than ".".

Availability System. An expression is available on entry to a block if it is available on exit from each predecessor of the block. An expression is available on exit from a block if it is locally available or if it is available on entry of the block and transparent in this block:

$$AVIN_i = \begin{bmatrix} \text{FALSE} & \text{if } i \text{ is the entry block} \\ \prod_{j \in Pred(i)} AVOUT_j & \text{otherwise} \end{bmatrix}$$

$$AVOUT_i = COMP_i + TRANSP_i \,.\, AVIN_i.$$

Anticipability System. An expression may be anticipated on exit of a block if it can be anticipated on entry of each successor of the block:

$$ANTOUT_i = \begin{bmatrix} \text{False} & \text{if } i \text{ is an exit block} \\ \prod_{j \in Succ(i)} ANTIN_j & \text{otherwise} \end{bmatrix}$$

$$ANTIN_i = ANTLOC_i + TRANSP_i \,.\, ANTOUT_i.$$

Partial Availability System. An expression is partially available on entry of a block if it is partially available on exit of at least one predecessor of the block:

$$PAVIN_i = \begin{bmatrix} \text{FALSE} & \text{if } i \text{ is the entry block} \\ \sum_{j \in Pred(i)} PAVOUT_j & \text{otherwise} \end{bmatrix}$$

$$PAVOUT_i = COMP_i + TRANSP_i \,.\, PAVIN_i.$$

2.2 Resolution of Boolean Systems

Allen and Cocke have introduced a method for redundancy elimination which is based on a partitioning of the program graph into subgraphs called intervals [2, 4, 5, 14].

An iterative approach to global-flow analysis which is easier to implement has also been developed by several authors [6, 9, 12]. Careful comparisons of the iterative approach with interval analysis appear both in [6] and [8]. Although there exist simple graph families such that the number of bit-vector operations is greater for the iterative approach than for the interval analysis [8], strong arguments in favor of iteration emerge from these comparisons.

An evaluation of the iterative approach applied to a study of 50 Fortran programs [10] has shown that the average upper bound of the number of iterations for solving the Boolean systems is 4.75 [6]. Our practical experience is that for well-structured programs written in the language LIS [7], numbering the nodes as they are created during the compilation process causes the number of iterations to be small (three in most cases).

The iterative process can give several solutions, depending on the initialization of the unknowns. For the Boolean systems presented earlier, the wanted solution is the largest solution for the systems involving the conjunction operator $\prod$ (availability and anticipability). The initialization value is then TRUE for all the unknowns. For the system of partial availability involving the disjunction operator $\sum$, the wanted solution is the smallest solution and the initialization value is FALSE for all the unknowns [6, 12].

In practice, it is possible to solve simultaneously the systems associated with 32 different expressions on a computer with words of 32 bits. Consequently, the time required to solve the Boolean systems is only a small fraction of the time required by the full optimization process.

2.3 Application to Redundant Expression Elimination

The construction of the local properties is made by a simple scan of every block. During this scan, the locally redundant computations are eliminated. It is possible to eliminate the first computation of an expression in a block i if it can be locally anticipated and is available on entry to the block i, i.e. if $ANTLOC_i \,.\, AVIN_i = \text{TRUE}$.

2.4 Application to Code Motion

Loop optimizations and suppression of certain partial redundancies involve moving computations from one point of the program to another. A computation of an expression can only be placed at a point where it may be anticipated since both safety and efficiency do not permit the introduction of a computation on a path where it was not present. A clear discussion of these safety and efficiency rules for code motion appears in [3, 14].

The removal of an invariant expression from a loop is only possible if this expression may be anticipated at the entry point of the loop, and if it is transparent in all the blocks of the loop. In classical techniques, removal

Fig. 1.

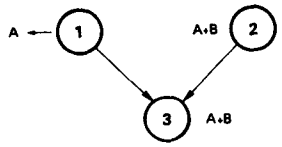
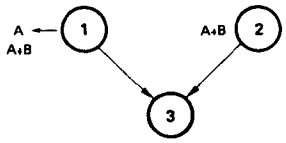


Fig. 2.



of loop invariants is performed loop by loop, from the innermost to the outermost.

Another form of loop optimization by code motion appears in [3] and [14] as "motion of nonloop constants." The notion of partial redundancy which embodies these cases has been presented in [12].

A computation of an expression is partially redundant in a block i if it may be locally anticipated and is partially available on entry to the block i, i.e. if $\text{ANTLOC}_i \cdot \text{PAVIN}_i = \text{TRUE}$. An optimization can be performed by moving the computation into predecessor blocks of i where the expression can be anticipated.

The computation of $A + B$ in node 3 of Figure 1 is partially redundant. This computation can be safely moved on exit of node 1 since the expression $A + B$ can be anticipated on exit of this node. Then, in Figure 2, the path (2, 3) contains one computation of $A + B$ instead of two as in Figure 1. The path (1, 3) always contains one computation.

An algorithm that globally performed this optimization on a block by block and expression by expression basis was tested in an earlier version of the LIS optimizer [12]. This algorithm being too costly, a new nongraphical algorithm handling all expressions at once was presented in [13]. The algorithm proposed in Section 3 of this paper is based on a generalization of the concept of partially redundant computation.

3. Global Utilization of Partial Redundancy Elimination

The redundant computations and the invariant computations of loops can be seen as particular cases of partially redundant computations.

Clearly, a redundant computation is also a partially redundant computation. Similarly, it can be shown that a loop-invariant computation is also a partially redundant computation.

Indeed, if a computation of an expression is invariant in a loop, this expression is transparent in all the blocks of the loop. Let i be any block of the loop containing a computation of the expression; since $\text{TRANSP}_i = \text{TRUE}$, then $\text{ANTLOC}_i = \text{COMP}_i = \text{TRUE}$, and then $\text{PAVOUT}_i = \text{TRUE}$. There exists a path P in the loop from i to i and all the blocks of P are transparent for the expression. The partial availability on exit of i then creates through P a partial availability on entry of i, and thus $\text{ANTLOC}_i \cdot \text{PAVIN}_i = \text{TRUE}$.

An algorithm which suppresses the partial redundancies of a program also eliminates the redundancies and the invariant computations of loops.

3.1 Algorithm for Suppression of Partial Redundancies

As in [3], the principle of the algorithm is to introduce new computations of the expression at points of the program chosen in such a way that the partially redundant computations become redundant and can hence be deleted.

The steps of the algorithm are as follows:

(a) Resolution of the Boolean systems for availability, anticipability, and partial availability.
(b) Determination of predecessors of the blocks containing the partial redundancies and where a new computation may be introduced. This involves the computation of the Boolean properties PPIN and PPOUT (Placement Possible on Entry and Placement Possible on Exit).
(c) Determination of a subset of these blocks on exit of which a computation must be inserted. These blocks satisfy the Boolean property INSERT.
(d) Insertion of new computations at the exit of the blocks satisfying INSERT = TRUE and suppression of the partially redundant computations which are now redundant.

3.1.1 Determination of PPIN and PPOUT. For the sake of clarity, an artificial constant term denoted CONST is introduced in the definition of PPIN . CONST_i is defined for every block i as

$$\text{ANTIN}_i \cdot [\text{PAVIN}_i + (\neg\text{ANTLOC}_i) \cdot \text{TRANSP}_i].$$

This constant term is TRUE for blocks containing a partial redundancy and for empty blocks where the expression can be anticipated. (A block is "empty" for an expression if it contains no computation nor modification of the expression.)

For every block i of the program, PPIN_i and PPOUT_i are then computed by:

$$\text{PPIN}_i = \begin{bmatrix} \text{FALSE if } i \text{ is the entry block, otherwise:} \\ \text{CONST}i \cdot \prod_{j \in \text{Pred}(i)} (\text{PPOUT}_j + \text{AVOUT}_j) \cdot (\text{ANTLOC}_i \\ + \text{TRANSP}_i \cdot \text{PPOUT}_i) \end{bmatrix}$$

$$\text{PPOUT}_i = \quad \text{FALSE if } i \text{ is an exit block, otherwise: } \prod_{k \in \text{Succ}(i)} \text{PPIN}_k.$$

This is a system of Boolean equations involving the conjunction operator $\prod$. The desired solution is the largest solution, and thus it is solved by the direct iterative method, starting with all TRUE's for the unknowns.

3.1.2 Determination of INSERT. For each block i of the program, INSERT_i is computed by:

$$\text{INSERT}_i = \text{PPOUT}_i \cdot \neg\text{AVOUT}_i \cdot (\neg\text{PPIN}_i + \neg\text{TRANSP}_i).$$

3.1.3 Insertion and Suppression of Computations. At the end of the algorithm, new computations are inserted on exit of the nodes satisfying INSERT = TRUE. Then the computations satisfying ANTLOC . PPIN = TRUE are redundant and may be deleted.

3.2 Proof of the Effectiveness of the Algorithm

The set of the deleted computations (satisfying ANTLOC . PPIN = TRUE) is only a subset of the set of the partially redundant computations (satisfying ANTLOC . PAVIN = TRUE). We must then prove that the transformation is nevertheless correct, and profitable. Theorem 1 proves that the deleted computations are redundant after the insertion of the new computations. In Theorem 2, we prove that none of the paths of the graph have been penalized. Conversely, all paths going through a node which contained a deleted computation have been optimized.

LEMMA 1. *After insertion of the new computations on exit of the blocks satisfying INSERT = TRUE, the new value of the availability on entry of any block for which* $PPIN_i$ *= TRUE will be* $AVIN'i$ *= TRUE.*

Preliminary Definition. A path going from the exit of a block i to the entry of a block j (denoted $]i, j[$) is an "empty path" for an expression if it does not contain any computation of the expression nor modification of its operands. The local Boolean properties for the blocks belonging to an empty path are TRANSP = TRUE and COMP = ANTLOC = FALSE.

The blocks i and j do not belong to $]i, j[$. The path $]i, j[$ is a single edge if $j \in \text{Succ}(i)$ and it then contains no blocks.

PROOF (Lemma 1). Let i be a block satisfying $PPIN_i$ = TRUE and let us assume that $AVIN'_i$ = FALSE. This can occur only if there exists:

(1) a block k such that $AVOUT_k = INSERT_k = TRANSP_k$ = FALSE (or if k is the entry block $AVOUT_k = INSERT_k$ = FALSE);
(2) an empty path $]k, i[$ such that $INSERT_m$ = FALSE for any block m belonging to $]k, i[$.

Intuitively, no computation already exists or will be inserted on the path $]k, i[$; then $AVOUT'_k$ = FALSE and this influence propagates through $]k, i[$, causing $AVOUT'_m$ = FALSE for every block m belonging to $]k, i[$ and then, $AVIN'_i$ = FALSE. $AVOUT_k = INSERT_k = TRANSP_k$ = FALSE implies $PPOUT_k$ = FALSE. (If k is the entry block, $PPIN_k$ = FALSE by definition and then $AVOUT_k = INSERT_k = PPIN_k$ = FALSE implies $PPOUT_k$ = FALSE.)

Let j be the successor of k in $]k, i[$, $PPOUT_k = AVOUT_k$ = FALSE implies $PPIN_j$ = FALSE. If $j = i$, we are led to a contradiction on the value of $PPIN_i$ else, $PPIN_j = AVOUT_j = INSERT_j$ = FALSE implies $PPOUT_j$ = FALSE. The same reasoning could then be applied to the block j and, by following the path from k towards i, we conclude that PPOUT = AVOUT = FALSE for the predecessor of i in $]k, i[$. Then $PPIN_i$ = FALSE, and we are led to a contradiction.

THEOREM 1. *After insertion of the new computations at the exit of the blocks where INSERT = TRUE, the first computation of the expression in any block i satisfying* $ANTLOC_i$ *.* $PPIN_i$ *becomes redundant.*

PROOF. These partially redundant computations obviously satisfy $PPIN_i$ = TRUE. According to Lemma 1, the new value of the availability is $AVIN'_i$ = TRUE. Since the insertion of new computations cannot change the value of $ANTLOC_i$, we will have $ANTLOC_i . AVIN'_i$ = TRUE and hence a redundancy.

LEMMA 2. *Let i be a block satisfying* $INSERT_i$ *= TRUE. Every path starting from the exit of i includes a computation which will be deleted.*

PROOF. $INSERT_i$ = TRUE implies $PPOUT_i$ = TRUE and then, $ANTIN_j$ = TRUE for every $j \in$ Succ(i). Hence there is a computation on every path starting from the exit of i. It remains to be proved that these computations will be deleted.

Let j be a block containing one of these computations and let us assume that it will not be deleted. According to Theorem 1, this implies $ANTLOC_j . PPIN_j$ = FALSE and since $ANTLOC_j$ = TRUE, $PPIN_j$ = FALSE. Since the computation located in j has created the anticipability of the expression on exit of i, there exists an empty path $]i, j[$.

$PPIN_j$ = FALSE implies PPOUT = FALSE for every predecessor block of j. Let k be the predecessor of j in $]i, j[$. If $k = i$, we are led to a contradiction on the value of $PPOUT_i$, else $PPOUT_k$ = FALSE, $TRANSP_k$ = TRUE, and $ANTLOC_k$ = FALSE implies $PPIN_k$ = FALSE. The same reasoning could then be applied to the block k and by following $]i, j[$ in the reverse order, we conclude that PPIN = FALSE for the successor of i in $]i, j[$. Then $PPOUT_i$ = FALSE, and we are led to a contradiction.

LEMMA 3. *Let i be a block satisfying* $INSERT_i$ *= TRUE. The paths starting from the exit of i cannot encounter another block satisfying INSERT = TRUE before encountering a block satisfying ANTLOC . PPIN = TRUE.*

PROOF. According to Lemma 2, every path starting from the exit of i encounters a block satisfying ANTLOC . PPIN = TRUE and this path is an empty path. Let j be one of these blocks and $]i, j[$ be the empty path.

Let k be a block of $]i, j[$ such that $INSERT_k$ = TRUE. Since $INSERT_k = PPOUT_k . \neg AVOUT_k . (\neg PPIN_k + \neg TRANSP_k)$, $INSERT_k$ = TRUE and $TRANSP_k$ = TRUE imply $PPIN_k$ = FALSE.

The proof of Lemma 2 shows that the value $PPIN_j$ = FALSE of a given block j propagates upwards through an empty path $]i, j[$ causing PPIN = FALSE for all the blocks of this path and finally $PPOUT_i$ = FALSE.

In the Lemma 3, $PPIN_k$ = FALSE causes $PPOUT_i$ = FALSE through $]i, k[$. This implies $INSERT_i$ = FALSE and we are led to a contradiction.

Fig. 3. Initial program.

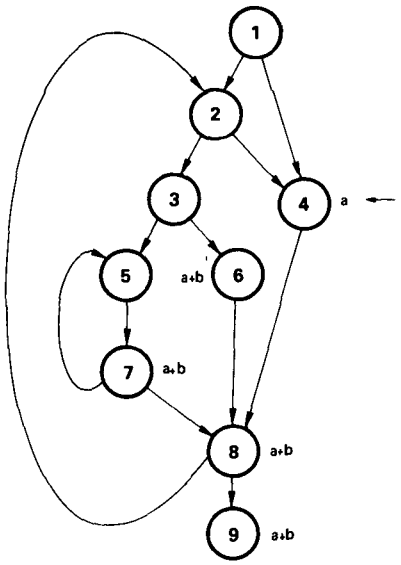
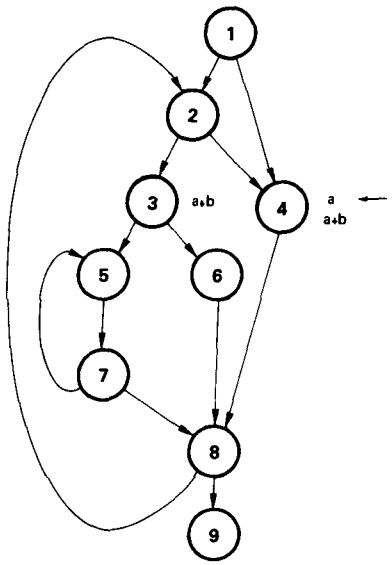


Fig. 4. Optimized program.



THEOREM 2. *At the end of the transformation, no path of the graph contains more computations of the expression than it contained before.*

PROOF. This is clear from Lemmas 2 and 3.

3.3 Application of the Algorithm to an Example

In this example (see Figures 3 and 4), an expression $a + b$ is computed in blocks 6, 7, 8, and 9. One of its operands is modified in block 4. The computation of the block 7 is invariant in the loop 5–7. The computation of the block 9 is redundant. The computations of the blocks 6 and 8 are partially redundant. For the sake of clarity, the correct initialization blocks of the loops have not been introduced.

Local Boolean Properties.

ANTLOC is	TRUE for nodes 6, 7, 8, 9;	FALSE elsewhere.
COMP is	TRUE for nodes 6, 7, 8, 9;	FALSE elsewhere.
TRANSP is	FALSE for node 4;	TRUE elsewhere.

Global Boolean Properties (obtained by resolution of Boolean systems).

ANTIN is	FALSE for nodes, 1, 2, 4;	TRUE elsewhere.
AVOUT is	TRUE for nodes 6, 7, 8, 9;	FALSE elsewhere.
PAVIN is	FALSE for node 1;	TRUE elsewhere.

Value of PPIN and PPOUT (obtained by resolution of Boolean systems).

PPIN is	FALSE for nodes 1, 2, 3, 4;	TRUE elsewhere.
PPOUT is	FALSE for nodes 1, 2, 8, 9;	TRUE elsewhere.

Computation of INSERT.

INSERT is TRUE for nodes 3, 4; FALSE elsewhere.

Insertion and Suppression of Computations.

ANTLOC . PPIN is TRUE for nodes 6, 7, 8, 9; FALSE elsewhere. Computations located in the blocks 6, 7, 8, and 9 are deleted. New computations are inserted at the exit of the blocks 3 and 4.

3.4 Discussion of the Algorithm

Having proved in Section 3.2 that the transformation does no harm, we can informally prove that this algorithm results in a program at least as efficient as the one produced by the successive application of the three classical techniques it aims to replace. This involves showing that if an optimization is performed by one of the classical techniques, it is also performed by this new algorithm.

(a) In the classical redundancy elimination, it is possible to eliminate the first computation of an expression in a block if ANTLOC . AVIN = TRUE. In the new algorithm, these blocks always satisfy ANTLOC . PPIN = TRUE, and the same computations are deleted.

(b) In the classical technique for invariants, a computation is removed from nested loops while it is both invariant in a loop and can be anticipated on entry of this loop. Let i be the initialization block of the outermost loop where the computation is placed in the classical treatment. In the new algorithm, the block i always satisfies $PPOUT_i$ = TRUE. If $INSERT_i$ = TRUE, a computation is introduced at the exit of the block i. But it may happen that the block i designated by the classical treatment is not the optimal point at which a new computation should be inserted since, for example, it may introduce a new partial redundancy which could be avoided. In this case, $INSERT_i$ is FALSE, and the computation is anticipated higher in the program. In any event, an invariant computation is removed by the new algorithm from at least as many loops as in the classical technique. Moreover, invariant computations can be removed in the same way from multientry loops although the classical technique is often limited to single-entry loops.

(c) The third classical technique which is subsumed by the new algorithm is the elimination of the partial redundancies which can still remain after having suppressed the redundancies and the invariant com-

Fig. 5.

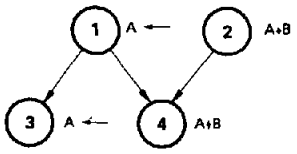


Fig. 6.

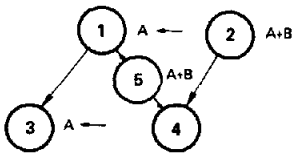


Fig. 7.

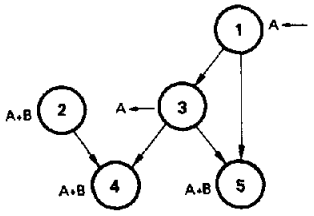
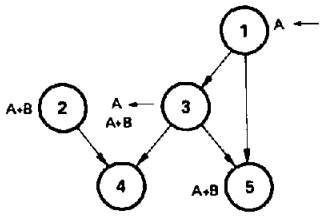


Fig. 8.



putations of loops. Since the new algorithm is based on this concept, it obviously performs this optimization.

The new algorithm can then replace these classical techniques without any loss of effectiveness in the resulting program. Nevertheless, some partially redundant computations may sometimes not be deleted. This appears in two cases:

(1) The expression cannot be anticipated at the points where new computations must be inserted. Doing this transformation would break the safety rule which prohibits the insertion of a computation on a path where it was not present. It is then impossible to suppress this partial redundancy without modifying the graph structure. This aspect of the problem is not discussed here.
The partial redundancy in node 4 of Figure 5 cannot be suppressed since the expression $A + B$ cannot be anticipated on exit of 1. It can only be suppressed by inserting a new node 5 on the edge (1, 4) with a computation of $A + B$ in it (see Figure 6).

(2) The partial redundancy can only be suppressed by introducing new partial redundancies. This can be done only if this transformation produces a run-time optimization by moving computations from a frequently used path to a less frequently used path. Execution frequency measurements are needed to guarantee the effectiveness of the transformation. The partial redundancy in node 4 of Figure 7 could be suppressed by inserting a computation of $A + B$ on exit of node 3. This transformation introduces a new partial redundancy in node 5 (see Figure 8). Although this transformation is "safe" since $A + B$ can be anticipated on exit of the node 3, it is not effective if the node 3 is more frequently executed than the node 4.

Fig. 9.

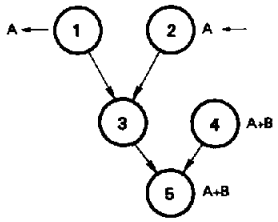


Fig. 10.

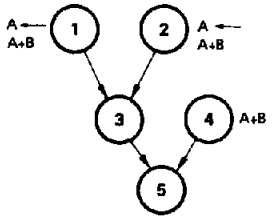
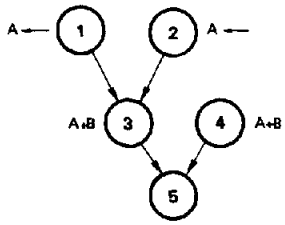


Fig. 11.



Moreover, this algorithm does not always allow the insertion of the minimal number of computations since it is sometimes possible to insert a computation in a common successor of the blocks designated by the algorithm.

The partial redundancy in node 5 of Figure 9 is eliminated by the insertion of two computations in nodes 1 and 2 (see Figure 10). A better solution, from a code size point of view, is the insertion of a unique computation in node 3 (see Figure 11). This can be achieved globally by applying, after the time-saving algorithm of this paper, a space-saving algorithm as "temporization" [12] which moves several computations in a common successor.

3.5 Experimental Results

Comparison has been made between two global optimizers developed for the language LIS [7]: the first one, described in [12] is using classical and independent techniques while the second one is based on the present algorithm. In both, the optimization phase is an optional and separate pass of the compiler.

In the first optimizer, the successive application of the classical techniques with the invariant moved outward loop by loop is a 7200 line program. This part of the optimizer can be reduced to 2500 lines if the new algorithm is used. In this case, given a description of the graph, a description of the expressions which can be optimized, and the value of the local Boolean properties for these expressions, the program which computes all the global Boolean systems, deletes, and inserts computations is a 800 line program. Our own experience is that it is considerably easier to implement than the classical methods since it completely eliminates the graphical problems for implicit loops, multientry loops, exit of nested loops, and avoids moving the same computation several times.

Although this technique does not have the control flow analysis costs of the usual methods, it must solve the additional data flow equations for PPIN and PPOUT. A meaningful theoretical evaluation seems to be very difficult and we have only experimentally measured the number of iterations for solving this system. This experiment has been made on a compilation of more than 50,000 lines of LIS programs with subroutines of up to 420 blocks. PPIN being initialized for each block with CONST (see Section 3.1.1) which is an upper value of the solution, the observed number of iterations has never exceeded three. Although it could be possible to have a slightly greater value with other tests, it appears that the maximum number of iterations is very small when optimizing actual programs as opposed to theoretical examples.

Experiments have shown that the time spent for global optimization is reduced by 30 to 60 percent when using the second optimizer. Moreover, the size of the global optimizer has been reduced by 35 percent and the optimizer can now run in the same space as the normal compilation process.

4. Conclusion

The redundant computations and the invariant computations of loops are particular aspects of partially redundant computations. In this paper, we have presented a new technique which allows a global elimination of partial redundancies, performing in a single algorithm what was done before by the successive application of several algorithms. This technique is based on a purely Boolean approach and hence permits a simultaneous treatment of all the expressions of a program.

The systems of Boolean equations are solved by a direct iterative method which is cheap and easy to implement. Experiments have shown that the cost of the algorithm is very nearly linear for well-structured programs; it depends mainly on the size of the program to be optimized and very slightly on its graphical structure.

This technique allows a simultaneous treatment of all the implicit and explicit loops of the program without any prior graphic identification, since it does not take into account the shape of the graph on which it is applied.

Our own experience is that it leads to a shorter optimizer which is easier to implement and runs faster. This technique is machine and language independent and can be applied to a wide class of optimizing compilers.

Acknowledgments. The authors benefited from discussions with J. D. Ichbiah. They would like to thank the referees for their careful reading of the manuscript and for a number of suggested improvements.

Received November 1976; revised September 1978

References

1. Aho, A. V., and Ullman, J. D. *The Theory of Parsing, Translation and Compiling, Vol. 2.* Prentice-Hall, Englewood Cliffs, N.J., 1973.
2. Allen, F. E. Control flow analysis. SIGPLAN Notices (ACM) 5, 7 (1970), 1–19.
3. Allen, F. E. and Cocke, J. A catalogue of optimizing techniques. In *Design and Optimization of Compilers*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 1–30.
4. Cocke, J. Global common subexpression elimination. SIGPLAN Notices (ACM) 5, 7 (1970), 20–24.
5. Goldberg, P. A comparison of certain optimization techniques. In *Design and Optimization of Compilers*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 31–50.
6. Hecht, M. S., and Ullman, J. D. A simple algorithm for global data flow analysis problems. *SIAM J. Comptng. 4* (1975), 519–532.
7. Ichbiah, J. D., Rissen, J. P., Heliard, J. C., and Cousot, P. The system implementation language LIS. Ref. Manual 4549 E/EN, CII-HB, Louveciennes, France, Dec. 1974.
8. Kennedy, K. A comparison of two algorithms for global data flow analysis. *SIAM J. Comptng. 5* (1976), 158–180.
9. Kildall, G. A. A unified approach to global program optimization. Conf. Rec. of ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, pp. 194–206.
10. Knuth, D. E. An empirical study of FORTRAN programs. *Software–Practice and Experience* (April 1971), pp. 105–134.
11. Lowry, E. S., and Medlock, C. W. Object code optimization. *Comm. ACM 12*, 1 (Jan. 1969), 13–22.
12. Morel, E., and Renvoise, C. Etude et réalisation d'un optimiseur global. Ph.D. Th., Université de Paris VI, Juin 1974 (English version available).
13. Morel, E., and Renvoise, C. A global algorithm for the elimination of partial redundancies. 2nd Int. Symp. on Programming, Paris, April 13–15, 1976, pp. 147–159 (edited by Dunod, Paris).
14. Schaefer, M. *A Mathematical Theory of Global Program Optimization.* Prentice-Hall, Englewood Cliffs, N.J., 1973.